Accelerating Machine Learning Applications through Optimized Tensor Decompositions

Emma Andrews and Prabhat Mishra
University of Florida, Gainesville, Florida, USA

***Abstract*—Tensors are fundamental in representing high-dimensional data across various application domains, including machine learning and quantum many-body simulation. Calculations on high-dimensional data represented as tensors are costly in terms of memory, power, and computation time. Tensor decomposition can reduce the dimensionality of the high-dimensional tensors by representing them as a sum or product of several low-dimensional tensors. In this paper, we present an automated framework for tensor decomposition for a series of operations in an application. Specifically, this framework automatically determines the beneficial tensor decomposition based on resource constraints. Experimental results demonstrate that our framework can drastically reduce memory requirements while providing accuracy comparable to state-of-the-art methods.**

***Index Terms*—Tensor decomposition, machine learning**

I. Introduction

Large-scale computations and big-data processing have become essential across a wide variety of disciplines, including machine learning and quantum many-body simulation. Unfortunately, these computational efforts face the curse of dimensionality where the computational complexity increases exponentially with the linear growth of dimensionality [1]. There are promising application-specific optimizations to reduce the dimensionality of data, such as neural network pruning in machine learning [2]. However, these techniques provide a reduction in dimensionality with a decrease in overall model accuracy or an increase in error rate.

The high-dimensional data is typically represented as tensors, or a collection of matrices, which have many rigorous properties. One such property is tensor decomposition – breaking down a higher-order tensor into several smaller-order tensors [3]. This decomposed format leads to significantly less data with minimal information loss. For example, Figure 1 showcases the reduction in memory requirement due to tensor decomposition. The left tensor of shape (60, 80, 100) containing 32-bit values requires approximately 1.92 MB of memory. In contrast, tensor decomposition produces the right result of one smaller-dimensional tensor and 3 matrices with 32-bit values, requiring approximately 2 KB of memory. These smaller-dimensional tensors can be multiplied and summed together in a specific order to reconstruct the original tensor.

While tensor decomposition is promising, it can be difficult to compute [3]. Tensors have many different ways to potentially calculate the decomposition, depending on the properties of the tensor, whereas their matrix counterparts use singular value decomposition (SVD) [4]. Furthermore,

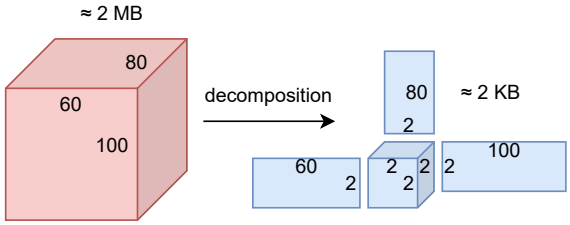

Fig. 1: Comparison of memory requirement between an original tensor (left) and its decomposed version (right).

tensor decomposition relies on the rank of a tensor, which is an NP-hard problem [5], causing the decomposition process to involve extra computation to approximate the rank. As a result, prior work focuses on using application-specific tensor decomposition to optimally compress a specific neural network architecture [6]–[9].

In this paper, we propose efficient tensor decomposition to accelerate machine learning applications based on constraints. Specifically, this paper makes the following contributions.

- Unlike existing methods that focuses on specific machine learning (ML) models, our framework can automatically determine the best-performing tensor decomposition format and rank and operation order for any ML model.
- Our framework can mitigate performance penalty by mapping common operations over certain dimension values to the best decomposition, without exhaustive searches for the format and rank pair or optimal contraction path.
- Evaluation using CNN, VGG19 [10], and ResNet18 [11] models demonstrate up to 2.9X reduction in memory requirement with negligible impact on accuracy.

The rest of the paper is organized as follows. Section II provides relevant background and surveys related efforts. Section III presents our proposed framework. Section IV presents experimental results. Finally, Section V concludes the paper.

II. Background and Related Work

A. Tensors

An order-n tensor is an n-dimensional multilinear array of data. A tensor can also be viewed as a collection of matrices, where a matrix is an order-2 tensor. Conversely, a vector is an order-1 tensor. We briefly outline a few operations on tensors.

1) Tensor Addition: Tensor addition is the process of adding a tensor with a scalar, vector, matrix, or other tensor. The slices of a tensor are used as matrices to perform element-

wise matrix addition. The collection of matrix additions across the slices results in the final value for the tensor addition.

2) Tensor Rank: Rank is an important measurement for both matrices and tensors as it is the minimum number of components required to express the matrix or tensor as a sum of those components [12]. Matrix rank can be easily computed, as it is the number of linearly independent rows or columns in the matrix. However, for tensors of order-3 or greater, determining the rank is an NP-hard problem [5]. Tensor decomposition (see Section II-B) is able to approximate the tensor rank while decomposing the original tensor as they need to determine a collection of components to represent the tensor.

3) Tensor Product: Tensor product involves multiplying a tensor by a scalar, vector, matrix, or other tensor. Often, the tensor is sliced into a matrix to perform the multiplication part and summed with the other slices to get the final product.

4) Tensor Contraction: Tensor contraction, also known as einsum, is an operation between tensors over a specified index [13]. As a result of the contraction, the size of the resulting tensor is the size of the remaining indices. For example, einsum for two order-2 tensors being contracted is `ik,kj->ij`, which means the two tensors are contracted over index `k`. This is equivalent to $C_{ij} = \sum_k A_{ik}B_{kj}$. For multiple terms, the order of operations is known as the contraction path.

B. Tensor Decomposition

Tensor decomposition breaks a tensor into smaller dimensional tensors to simplify calculations [3]. There are three popular tensor decomposition methods: canonical polyadic (CP) [14], Tucker [15], [16], and tensor train (TT) [17].

1) Canonical Polyadic (CP) Decomposition: CP decomposition [14] aims to decompose a high-order tensor into a sum of rank-1 tensors. A rank-1 tensor is represented by a factor matrix, where each column is a vector of the rank-1 tensor, such as $\mathbf{A} = [\,a_1\ a_2\ \cdots\ a_R\,]$. For an order-3 tensor $\mathcal{X} \in \mathbb{R}^{I\times J\times K}$, we can express its CP decomposition as the outer product of each mode from the rank-1 tensors, such that

$$\mathcal{X} \approx \sum_{r=1}^{R} \lambda_r a_r \otimes b_r \otimes c_r, \tag{1}$$

where R is the rank of the tensor and $\lambda \in \mathbb{R}^R$ is a weight vector, assuming the columns of each factor matrix are normalized.

For an order-n tensor $\mathcal{X} \in \mathbb{R}^{I_1\times I_2\times\cdots\times I_N}$, the CP decomposition can be expressed as

$$\mathcal{X} \approx \sum_{r=1}^{R} \lambda_r a_r^{(1)} \otimes a_r^{(2)} \otimes \cdots \otimes a_r^{(N)}, \tag{2}$$

where $a^{(n)}$ is a column vector for the factor matrix $A^{(n)} \in \mathbb{R}^{I_n\times R}$. For example, Figure 2 shows that tensor $\mathcal{X}$ is represented by the sum of the rank-1 tensors, displayed in vector form for each factor matrix.

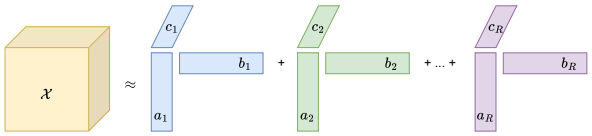

Fig. 2: CP decomposition for an order-n tensor $\mathcal{X}$.

2) Tucker Decomposition: Tucker decomposition [15], also known as the higher-order singular value decomposition [16], expresses an order-d tensor as a smaller order d core tensor $\mathcal{G}$ and d factor matrices. For an order-3 tensor $\mathcal{X} \in \mathbb{R}^{I\times J\times K}$, the corresponding Tucker decomposition is

$$\mathcal{X} \approx \sum_{p}^{P}\sum_{q}^{Q}\sum_{r}^{R} g_{pqr} a_p \otimes b_q \otimes c_r, \tag{3}$$

where a_p, b_q, and c_r are the modes for each of the factor matrices $\mathbf{A} \in \mathbb{R}^{I\times P}$, $\mathbf{B} \in \mathbb{R}^{J\times Q}$, and $\mathbf{C} \in \mathbb{R}^{K\times R}$, similar to CP decomposition. g_pqr is an entry in the core tensor $\mathcal{G} \in \mathbb{R}^{P\times Q\times R}$.

An order-n tensor $\mathcal{X} \in \mathbb{R}^{I_1\times I_2\times\cdots\times I_N}$ can be decomposed as

$$\mathcal{X} \approx \sum_{r_1=1}^{R_1}\sum_{r_2=1}^{R_2}\cdots\sum_{r_N=1}^{R_N} g_{r_1 r_2 \ldots r_N} a_{r_1}^{(1)} \otimes a_{r_2}^{(2)} \otimes \cdots \otimes a_{r_N}^{(N)}, \tag{4}$$

where $a^{(n)}$ is a column vector for the factor matrix $A^{(n)} \in \mathbb{R}^{I_n\times R}$. For example, Figure 3 shows a sample Tucker decomposition for a tensor $\mathcal{X}$, consisting of core tensor $\mathcal{G}$ and factor matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$. Each factor matrix corresponds with a specific mode of the core tensor $\mathcal{G}$.

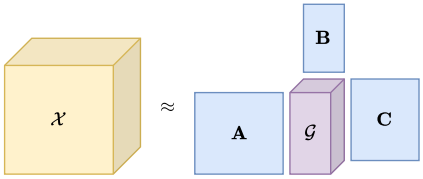

Fig. 3: Tucker decomposition for an order-n tensor $\mathcal{X}$.

3) Tensor Train (TT): Tensor train [17] decomposes an order-n tensor into n order-3 tensors, which are chained together through a product operation, typically matrix multiplication. Formally, given an order-n tensor $\mathcal{X} \in \mathbb{R}^{I_1\times I_2\times\cdots\times I_N}$, TT is approximated by

$$\mathcal{X} \approx \mathbf{G}_{i_1}^{(1)}\mathbf{G}_{i_2}^{(2)}\ldots\mathbf{G}_{i_N}^{(N)}, \tag{5}$$

where $\mathbf{G}_{i_j}^{(j)}$ is an $r_{j-1}\times r_j$ matrix. Figure 4 shows an example of a tensor being decomposed into TT format to the smaller factor matrices. Each line between tensors indicates the shared index between the two tensors. TT is often used for calculating the matrix product state in quantum computing.

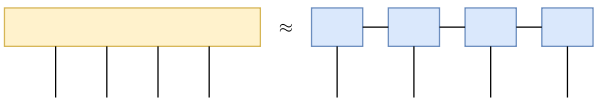

Fig. 4: An original tensor and its corresponding tensor train.

4) Other Variants: There are variations of the major three decomposition formats. One such variant is the non-negative versions of CP and Tucker, where the tensor to be decomposed must contain only non-negative values. Other variants include PARAFAC2, CANDELINC, INDSCAL, and DEDICOM [3].

C. Related Work

Prior work focuses on compressing a specific model architecture by using tensor decompositions, tailoring the tensor decomposition method to fit the mathematical properties of the model [6]–[9]. Other work focuses on determining the best decomposition format for a specific model architecture and the given number of parameters. HEAT is an automated framework for producing tensor decomposition specifically for transformer architectures and their attention mechanisms [18]. Applicability is a major limitation of prior efforts since they focus on specific aspects of the model architecture that may not be shared across ML models. Our work addresses this problem by providing a universal framework for optimized tensor decomposition for a wide variety of machine learning models and associated applications.

III. OTD: Optimized Tensor Decomposition for Machine Learning Applications

We present Optimized Tensor Decomposition (OTD), a framework for optimizing a list of tensor operations by using tensor decompositions. OTD picks a format and rank pairing that would achieve the best performance given several tradeoffs, such as memory cost vs. computational time. Our primary application is machine learning, such as optimizing the calculations within the layers of a neural network.

Given a list of operations, such as the layers of a neural network, our objective is to optimize these operations for the best possible accuracy, memory usage, and computational time under various resource constraints. For each operation, we take each tensor operand and compute the possible tensor decomposition formats, calculating hardware metrics such as run time and memory. From these options, we analyze the operation, estimating the best tensor decomposition format and contraction path for the given operation. While this is done at an operation level, considerations are taken to ensure that the remaining operations, whether sequential or parallel operations, also have effective decompositions and contraction paths given their operands and the overall hardware constraints. These hardware constraints are based on the architecture of the hardware that is being used to calculate the list of operations.

Figure 5 provides an overview of our framework. First, the list of operations is broken into individual operations, and then each operation's operands. These operands are decomposed into different decomposition format and rank pairs, calculating several cost metrics related to the decomposition process and the resulting decomposed tensors. Next, different combinations of the format and rank pairs for each operand are tested within the current operation. Cost metrics are also calculated at this level. Finally, all the operations are put together, optimizing for any intermediate calculations between the individual operations and retrieving the best possible order of operations with the most optimized tensor operands based on their format and rank pair and the cost metrics.

As more operations are examined, OTD learns consistencies between tensor shapes and their related operations, and when it encounters a learned consistency, it will automatically determine the best decomposition format and rank without having to reproduce the costs, saving performance.

A. Optimization of Cost Metrics

We measure cost in three distinct scenarios: (1) performing the decomposition and its resulting tensors, (2) individual tensor operations, and (3) a holistic view of all operations. Each operation must factor in the cost of performing the operation between two tensors, such as the optimal contraction path, and determine what decomposition formats are the best performing within the constraints of the operation. The choice between decomposition formats must also consider the cost of decomposing the original tensor into each of the formats, as well as the storage requirements to store each of the decomposed formats. The operations must be mapped in an optimized order given all the contraction paths and individual operation costs. Algorithm 1 outlines the major steps in our framework. The remainder of this section describes these steps in detail.

Algorithm 1: Optimization of All Cost Metrics

Data: Z_u: list of operations with tensor operands, F: format set, R: rank set
Result: Z_d, optimized operations using decompositions

for $o \in Z_u$ **do**
  for $p \in o$ **do**
    $d_g = \texttt{memory}(p)$;
    for $(f, r) \in F, R$ **do**
      $x = \texttt{decompose}(p, f, r)$;
      $d_p(f, r) =$ $\texttt{time}(x) + \texttt{memory}(x) + \texttt{error}(x)$;
      if $d_p(f, r) > d_g$ **then**
        $d_p(f, r) = d_g$;
      end
    end
  end
  $o_d = \min_{i=1,\ldots,q;j=1,\ldots,m}(d_i((f, r)_j) + \texttt{time}(o))$;
end
$Z_d = \texttt{reduction}(o_d)$;

1) Tensor Decomposition Cost: Each of the three formats calculates the decomposed tensor in different ways (CP, Tucker, and TT), as detailed in Section II-B, and are based on the tensor's approximate rank value. Since our objective is to reduce the tensor based on hardware constraints, we can instead choose the rank for decomposition. Therefore, given a tensor operand p from an operation o, we must determine the best tensor decomposition based on the possible format and

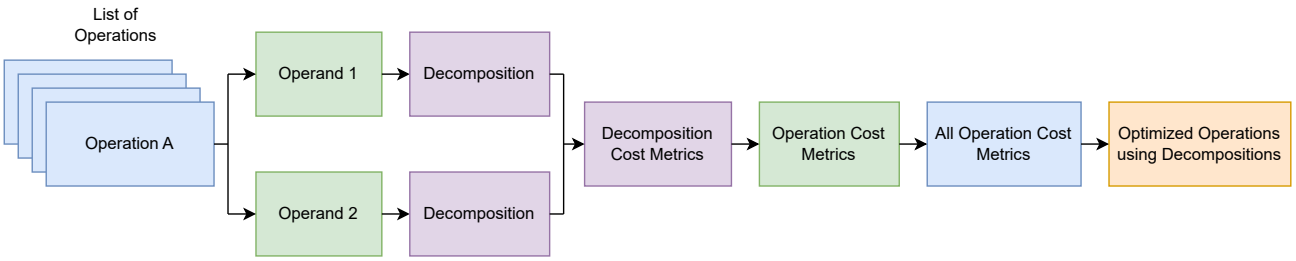

Fig. 5: OTD architecture. Cost metrics are gathered at three different levels: decomposition, individual operation, and all operations. Based on the metrics and given constraints, an optimized order of operations using decomposition is produced.

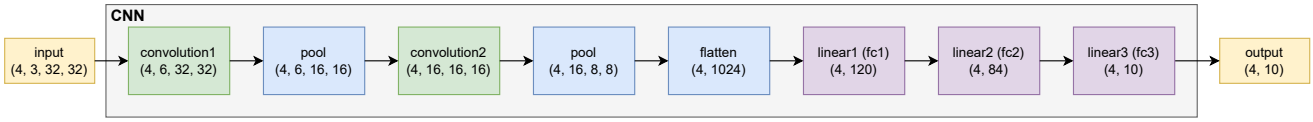

Fig. 6: Shapes for each layer are the resulting output after that layer has calculated its results. This is the input to the next layer, except for the final layer where it contains the probability percentages for each input image as to the class label it belongs to.

rank (f, r) pairs from the overall format F and rank R sets. The format set F contains the three popular formats. The rank set R contains the rank values to test, which is any positive integer. However, we can reduce the total number of members of the rank set, and thus overall pairs, by testing a subset of ranks. For example, we can omit testing rank 3 if we test rank 2 given a large original tensor since there will be a negligible cost difference between the two results.

Once the set of format and rank pairs (f, r) are determined, we decompose the tensor p with each pair value. Taking the result of the decomposition operation, we examine the time it took to decompose the tensor, the memory requirement to store all components of the decomposed tensor, and the error rate induced from decomposition, as shown in Line 6 of Algorithm 1. The error rate is calculated by taking the mean squared error between the reconstructed version of the decomposed tensor and the original tensor.

The original tensor may perform better than its decomposed version. Therefore, we check the resulting cost metrics after decomposing against the original tensor's memory cost. This is prevalent when the rank chosen results in components that consist of more total maximum memory than the original tensor. For example, given a (2, 3, 4) size tensor, decomposing in Tucker format with rank 9 results in components totaling memory of 3296 bytes, which is more total memory than the original tensor of 3000 bytes (see Table I). If the original tensor performs better, we omit the specific format and rank pair from consideration (Line 8).

2) Individual Operation Cost: For each individual operation o, we gather the tensor decomposition metrics $d_p(f, r)$ for each tensor operand p. With q total operands and m pairs for each operand, Line 12 chooses the most suitable pair for every operand given the computation time of the operation using the decomposed operand variants.

Each operation is computed with the corresponding einsum between all operands. This einsum is optimized to ensure the maximal performance given the necessary contraction paths for the desired result. When the einsum operands are decomposed in a specific format, the einsum also optimizes the intermediate steps performed between the factor matrices of the decomposed tensor in the full operation. For example, an original order-4 tensor decomposed using CP format will have the einsum `ar,br,cr,dr->abcd` to be reconstructed to the original shape. If this decomposed tensor were to be contracted with a tensor of `ebcd` to get a result of `eacd`, the einsum would be `ar,br,cr,dr,ebcd->eacd`, which can then be optimized further. The resulting optimized operation is denoted as o_d.

Optimizing the contraction path of each einsum is a necessary step due to the potential differences in speed based on the contraction path taken to get to the same result [19]. With multiple components, a new contraction path may be available that can result in faster execution time in comparison to the optimal contraction path for the original tensor. Therefore, we aim to include this beneficial speedup where possible.

3) Total Operation Cost: Along with optimizing at the individual operation level, we analyze all operations with their order of operation to verify that they are being executed in an optimized order, with no potential reductions across different individual operations. Any extra computational cost as a result of performing this operation is added onto the overall cost metric. After this cost is calculated, Algorithm 1 returns the optimized list of operations with the best decomposed tensor operands given the hardware constraints.

B. Application: Machine Learning

Machine learning utilizes high-order tensors as the main data structure for different operations. Due to this, we provide an example of OTD being used to optimize the performance of a basic convolutional neural network (CNN). The CNN consists of several layers, each containing an operation between tensors. For example, the linear layer is computed as the multiplication between the weight tensor of the layer and the input tensor. The 2D convolution layer is a specific sliding

window multiplication between the weight tensor and the input tensor, using a small matrix called a kernel as the window.

Figure 6 contains a breakdown of an example CNN, displaying the layers that an input is calculated to reach the classification prediction. In this example, the CNN contains convolutional and linear layers as described previously, with the layers acting as the set of operations to optimize in Algorithm 1. The pool layer reduces the shape of the input tensor by taking the max of the inputs. The flatten layer reshapes the tensor, in this case, by removing two dimensions, reducing it from an order-4 tensor to a matrix. As these two layers do not involve any direct matrix or tensor multiplication, tensor decomposition is not applicable and will function as normal. For the convolution and linear layers, each layer contains a weight matrix or tensor that is multiplied by the input matrix or tensor. This weight tensor is a candidate for tensor decomposition, with our framework determining the appropriate format and rank to decompose with for the best accuracy, memory requirements, and computational time.

We do not decompose the input to each layer due to the drastic increase in time. The weight tensors can remain decomposed after initial decomposition, however, the inputs would have to be decomposed again at the beginning of each layer. This results in an exponential time cost during training as these layers are calculated over 12,000 times in the case of CIFAR-10 with batches of 4 images in one training epoch. Moreover, due to each layer feeding its result into the next layer, there is negligible optimization that can be done at a total operation level.

IV. Experiments

In this section, we present experimental results using our framework, OTD. First, we look at finding effective tensor decompositions based on Algorithm 1. We then examine using these selections for a CNN, showcasing its impact on machine learning applications. Finally, we briefly examine our framework being used with other machine learning architectures.

A. Experimental Setup

All experiments were run on Linux using an Intel 13900K, NVIDIA RTX 4090, and 64GB of RAM. The framework uses Python v3.11, PyTorch v2.2, tensorly v0.8.2, and opt_einsum v3.3 [20], [21]. We use the traditional PyTorch implementations of the models as our baseline. We adjust these models with our decomposition layers to provide a comparison between the traditional approach and our decomposed approach.

B. Selection of Tensor Decomposition Parameters

With tensor decompositions, there are many possible format and rank pairings, each potentially giving a different decomposition in terms of component size and number of overall components. Tensors decomposed at rank 2 result in the smallest memory footprint. Table I shows a variety of different format and rank pairings and the corresponding memory, time taken, and overall error rate versus the original tensor. In this specific case, our selection parameters indicate that the smallest memory savings is through CP with rank 2, resulting in 5 tensors of varying sizes totaling 192 bytes. This contrasts with the original tensor requiring 3,000 bytes.

While we can get the largest reduction in memory, this does result in large error rates between the original tensor, meaning information that was lost during the decomposition process. For more error-sensitive applications, one may wish to have a decomposition that has less error rate between the original tensor, at the tradeoff of requiring more memory, however, this memory will still be less than the total memory of the original tensor. For example, Tucker rank 4 results in 5 component tensors totaling 1,124 bytes of memory, but with a much lower error rate compared to CP rank 2.

TABLE I: Decompositions for a (2, 3, 4) tensor.

Format	Rank	Memory (bytes)	Time (s)	Error Rate
Original	-	3000	-	-
CP	2	[8, 80, 24, 40, 40]	0.482	0.0421
CP	4	[16, 160, 48, 80, 80]	0.450	0.0838
CP	7	[28, 280, 84, 140, 140]	0.409	0.122
CP	9	[36, 360, 108, 180, 180]	0.382	0.146
Tucker	2	[64, 80, 24, 40, 40]	0.476	0.00345
Tucker	4	[768, 160, 36, 80, 80]	0.385	0.00175
Tucker	7	[2100, 280, 36, 100, 100]	0.213	0.00211
Tucker	9	[2700, 360, 36, 100, 100]	0.111	0.00223
TT	2	[80, 48, 80, 40]	0.478	0.000545
TT	4	[160, 192, 320, 80]	0.424	0.000239
TT	7	[280, 588, 700, 100]	0.337	0.000332
TT	9	[360, 972, 900, 100]	0.279	0.00029

C. Explorations of Tensor Decompositions for CNNs

We provide results for the CNN described in Section III-B, creating the CNN in PyTorch to classify CIFAR-10 images [22]. Images can be represented as a tensor of size (H, W, C), where H is the height of the image, W the width, and C the number of channels. For CIFAR-10, these tensors are of shape (32, 32, 3), with 3 channels to represent the color. Additionally, training and testing data are often batched together in a single call to the neural network to save time at the cost of working with higher-dimensional data. Batching of data results in multiple (32, 32, 3) images stacked together. If 4 images were batched together, the resulting batched tensor would be of shape (4, 32, 32, 3).

The 2D convolutional layer takes an input consisting of 3 channels and produces an output with 6 channels. The convolution operation is performed with a kernel size of 5. Therefore, the input of size (4, 3, 5, 5), using the batch of 4 inputs as described earlier, is convoluted with the layer's weight tensor of size (6, 3, 5, 5). As the neural network stores its layers' weights in full form, we do not decompose the weight tensor. The resulting output is of size (4, 6, 5, 5), which is used as input to the next layer of the CNN.

The linear layer takes an input consisting of either a vector or matrix, depending on if the original input is batched. If batched, the input matrix will be of size (N, I), where N is the batch size and I the number of in channels (features) of the layer, with the output matrix being of size (N, O), where

TABLE II: Results for CNN. Italicized layers are decomposed layers. Bold values are the chosen best variant.

			Classification Time (μs)						Layer Memory (KB)					
Framework	Accuracy	Training Time (s)	conv1	conv2	fc1	fc2	fc3	total	conv1	conv2	fc1	fc2	fc3	total
Traditional	65.47%	357.75	29.1	24.3	17.4	8.8	7.9	87.5	4.8	51.2	983.0	40.3	3.4	1082.7
OTD-Opt	61.10%	615.58	*635*	41.7	15.7	8.3	8.3	709	*1.2*	51.2	983.0	40.3	3.4	1079.1
OTD-Opt	**65.29%**	452.08	30.3	25.0	*162*	15.0	8.3	**240.6**	4.8	51.2	*277.5*	40.3	3.4	**377.2**
OTD-Opt	68.17%	447.17	28.8	24.6	17.2	8.6	*160*	239.2	4.8	51.2	983.0	40.3	*2.3*	1081.6
OTD-Opt	68.11%	598.82	31.7	25.5	*166*	*114*	*124*	461.2	4.8	51.2	*277.5*	*26.2*	*2.3*	362.0
OTD-Opt	54.88%	1175.98	*455*	*898*	*148*	*116*	*252*	1869	*1.2*	*12.1*	*277.5*	*26.2*	*2.3*	319.3
OTD-Low	59.85%	659.04	*592*	38.6	15.5	8.8	7.4	662.3	*0.2*	51.2	983.0	40.3	3.4	1078.1
OTD-Low	50.71%	457.11	32.2	26.5	*164*	15.0	8.6	246.3	4.8	51.2	*17.3*	40.3	3.4	117.0
OTD-Low	63.71%	467.81	29.1	24.6	18.4	8.3	*166*	246.4	4.8	51.2	983.0	40.3	*0.8*	1080.1
OTD-Low	52.86%	622.69	32.9	25.3	*167*	*116*	*123*	464.2	4.8	51.2	*17.3*	*1.6*	*0.8*	75.7

TABLE III: Comparison between traditional models and best-performing OTD models.

	Accuracy			Classification Time (μs)			Layer Memory (KB)		
NN	Traditional	OTD	Loss	Traditional	OTD	Increase	Traditional	OTD	Reduction
CNN	65.47%	65.29%	0.18%	87.5	240.6	153.5	1,082.7	377.2	2.9x
VGG19	80.88%	80.32%	0.56%	39.6	217	177.4	478.3 MB	411.2 MB	1.2x
ResNet18	74.54%	74.44%	0.10%	18.4	154	135.6	20.5	12.6	1.6x

O is the number of out channels (features). The weight matrix, regardless if the input is batched, is always of size (I, O). If the input is not batched, the input and output matrices become vectors, removing the first dimension of size N.

For our experiments, we analyzed different rank values for each of the formats to determine its effect on the total memory, training time, and accuracy. Table II consists of several of these experiments, following the selection criteria outlined in Section IV-B. Along with results for the most memory-saving selections and a middle point in memory savings, we choose different combinations of CNN layers to use the equivalent decomposed layer. The CNNs with decomposed fc1, fc2, and fc3 (CNN-fc123) and decomposed fc3 (CNN-fc3) performed the best in terms of accuracy, with CNN-fc123 only 0.06% less accurate. CNN-fc123 also further reduced the total amount of required memory, needing a total of 362 KB in comparison to CNN-fc3's 1.1 MB. However, CNN-fc123 suffers from much longer training and inference time. To get the benefits of drastically reduced memory without a large time penalty, we tradeoff some accuracy with decomposing the fc1 layer.

We have also included results from using decompositions that provide the smallest memory footprint. In every configuration comparison, the smallest memory footprint performs worse in time taken and accuracy. However, the smallest memory footprint is able to drastically reduce the memory, such as the fc1 layer going from 983 KB originally to only needing 17 KB, a 98% decrease.

In general, as the memory decreases, the accuracy also decreases. However, we see the opposite trend with time and accuracy. When considered together, memory and time are tradeoffs in terms of overall accuracy. Typically, as the memory decreases, the timing increases due to needing more time at the decomposed layers to compute the corresponding einsum. This einsum, while operating between much smaller tensors, requires several multiplication operations compared to just one in the traditional layers.

There are some cases where the tensor decomposition leads to better accuracy, despite having much smaller weight tensors in the layers. This is likely due to the tensor decomposition, through the reduction in dimensionality, reducing the sparsity of the original weight tensor that can negatively impact the accuracy of the model [23].

D. Tensor Decomposition Across Neural Networks

There are several architectural improvements on the basic CNN for better accuracy and performance in generic cases. We have tested OTD with two of these architectures, VGG19 [10] and ResNet18 [11]. We compare the best-performing decomposition combination with the traditional architecture in Table III. For each model, our best performing decomposition variant has comparable accuracy with less memory requirements. As a tradeoff, time is increased for classifying samples. This classification time is short enough (a few μs) to be of any practical concern.

V. Conclusion

Tensor decomposition is a promising technique for breaking down a high-order tensor into smaller tensors and matrices. While tensor decomposition is beneficial for machine learning applications with high-dimensional data, there are cost and performance trade-offs that must be considered for the most appropriate format and rank. We proposed a framework that can automatically find the optimal tensor decomposition for all tensors as well as the optimal order of operations for given a list of operations. The operation order is at the individual operation layer, optimizing the decomposed form of the tensor by the other operand, and also at the combined operation layer, ensuring that operations cannot be reduced further for even

better performance. Extensive experimental evaluation using diverse machine learning models demonstrated significant improvement in memory requirement with negligible impact on prediction accuracy.

REFERENCES

[1] I. V. Oseledets and E. E. Tyrtyshnikov, "Breaking the Curse of Dimensionality, Or How to Use SVD in Many Dimensions," *SIAM Journal on Scientific Computing*, vol. 31, no. 5, pp. 3744–3759, Jan. 2009.

[2] D. Blalock, J. J. Gonzalez Ortiz, J. Frankle, and J. Guttag, "What is the State of Neural Network Pruning?" *Proceedings of Machine Learning and Systems*, vol. 2, pp. 129–146, Mar. 2020.

[3] T. G. Kolda and B. W. Bader, "Tensor Decompositions and Applications," *SIAM Review*, vol. 51, no. 3, pp. 455–500, Aug. 2009.

[4] D. Kalman, "A Singularly Valuable Decomposition: The SVD of a Matrix," *The College Mathematics Journal*, Jan. 1996.

[5] C. J. Hillar and L.-H. Lim, "Most Tensor Problems Are NP-Hard," *J. ACM*, vol. 60, no. 6, pp. 45:1–45:39, Nov. 2013.

[6] Y. Pan, J. Xu, M. Wang, J. Ye, F. Wang, K. Bai, and Z. Xu, "Compressing Recurrent Neural Networks with Tensor Ring for Action Recognition," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, pp. 4683–4690, Jul. 2019.

[7] A.-H. Phan, K. Sobolev, K. Sozykin, D. Ermilov, J. Gusak, P. Tichavský, V. Glukhov, I. Oseledets, and A. Cichocki, "Stable Low-Rank Tensor Decomposition for Compression of Convolutional Neural Network," in *Computer Vision – ECCV 2020*, 2020, pp. 522–539.

[8] A. Tjandra, S. Sakti, and S. Nakamura, "Tensor Decomposition for Compressing Recurrent Neural Network," in *2018 International Joint Conference on Neural Networks (IJCNN)*, Jul. 2018, pp. 1–8.

[9] M. Yin, S. Liao, X.-Y. Liu, X. Wang, and B. Yuan, "Compressing Recurrent Neural Networks Using Hierarchical Tucker Tensor Decomposition," May 2020.

[10] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," Sep. 2014.

[11] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[12] J. B. Kruskal, "Three-way arrays: Rank and uniqueness of trilinear decompositions, with application to arithmetic complexity and statistics," *Linear Algebra and its Applications*, vol. 18, no. 2, pp. 95–138, Jan. 1977.

[13] D. A. Matthews, "High-Performance Tensor Contraction without Transposition," *SIAM Journal on Scientific Computing*, vol. 40, no. 1, pp. C1–C24, Jan. 2018.

[14] F. L. Hitchcock, "The Expression of a Tensor or a Polyadic as a Sum of Products," *Journal of Mathematics and Physics*, vol. 6, no. 1-4, pp. 164–189, 1927.

[15] L. R. Tucker, "Some mathematical notes on three-mode factor analysis," *Psychometrika*, vol. 31, no. 3, pp. 279–311, Sep. 1966.

[16] L. De Lathauwer, B. De Moor, and J. Vandewalle, "On the Best Rank-1 and Rank-(R1 ,R2 ,. . .,RN) Approximation of Higher-Order Tensors," *SIAM Journal on Matrix Analysis and Applications*, vol. 21, no. 4, pp. 1324–1342, Jan. 2000.

[17] I. V. Oseledets, "Tensor-Train Decomposition," *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2295–2317, Jan. 2011.

[18] J. Gu, B. Keller, J. Kossaifi, A. Anandkumar, B. Khailany, and D. Z. Pan, "HEAT: Hardware-Efficient Automatic Tensor Decomposition for Transformer Compression," Nov. 2022.

[19] E. Meirom, H. Maron, S. Mannor, and G. Chechik, "Optimizing Tensor Network Contraction Using Reinforcement Learning," in *Proceedings of the 39th International Conference on Machine Learning*. PMLR, Jun. 2022, pp. 15 278–15 292.

[20] J. Kossaifi, Y. Panagakis, A. Anandkumar, and M. Pantic, "TensorLy: Tensor Learning in Python," *Journal of Machine Learning Research*, vol. 20, no. 26, pp. 1–6, 2019.

[21] D. G. A. Smith and J. Gray, "Opt_einsum - A Python package for optimizing contraction order for einsum-like expressions," *Journal of Open Source Software*, vol. 3, no. 26, p. 753, Jun. 2018.

[22] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," 2009.

[23] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, "Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks," *Journal of Machine Learning Research*, vol. 22, no. 241, pp. 1–124, 2021.